Visual Special Relativity

Victor Bohlin

February 1, 2010
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Thomas Johansson
Examiner: Fredrik Georgsson

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Albert Einstein predicted in 1905 with his paper on special relativity the significant visual effects of objects moving close to the speed of light. Einstein could not see them then but with todays graphical hardware it is possible to produce these effects in real time in a computer simulation.

In this thesis the implementation, results and comparison of two techniques for doing this are presented. The two techniques differ substantially in how they execute the task. One alters the vertex positions of the objects and the other performs the effects as a post process step using a cube map. The results of the two techniques are generally the same, both having different advantages and disadvantages.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Einstein's theory of special relativity incorporates the principle that there is no well-defined state of rest and that the speed of light is the same for all observers regardless of their relative motion. The theory predicts that the apparent visual forms of objects moving close to the speed of light (in relation to the observer) will change. One example is a contraction in the direction of motion. However, when predicting the apparent forms of such objects one has to also take into account that the speed of light is finite. This fact leads to additional changes perhaps more prominent than the contraction in length. Examples of these changes include aberrational effects, a compression of light rays into a cone in front of the moving observer in the forward direction and a Doppler shift in the wavelengths of the light rays.

These effects are rarely[1] observed in real life but can be visualized in a computer simulation where either the objects are astronomically big and the observer is moving close to the speed of light or where the speed of light is simply scaled down close to the speed of the observer.

The main task of this Master's Thesis project is to investigate and implement different techniques of visualizing these effects in a simulation where the user can move relativistically in an environment consisting of several different objects. The application is implemented using the Microsoft XNA platform which is designed to facilitate creation of games and multimedia products. The actual effects are implemented using HLSL shaders.

The application resulting from this project should let the user experience what it is to travel close to the speed of light. It may be used together with a hardware rig to produce a highly immerse experience where the user is able to manipulate his/her own speed as well as the speed of light.

The project was performed at Coldwood Interactive which is a game developing company with 14 employees situated in central Umeå. They have produced games such as Ski-Doo Snowmobile Challenge, Skiracing 2005 featuring Hermann Maier, Skiracing 2006, Freak Out - Extreme Freeride and OnGolf.

In chapter two the visual effects of relativistic travel are presented with intuitive descriptions together with the theoretical equations that lie as a foundation for their implementation. Chapter three contains the problem description, what features that should be included in the final application and the priority of these. This chapter also contains a section briefly describing what work has already been done by others related to visualizing the effects of special relativity. In chapter four a conceptual graphical pipeline is described to give an understanding of the functionality and power of programmable shaders. In chapter five the two

[1] The Doppler shift is sometimes visible when observing moving stars.

implemented techniques used to visualize the aberrational effect are compared, both theoretically with regards to performance and practically with regards to performance, visual quality and usability. In chapter six the process of how the work was executed is presented and compared to the preliminary schedule. Chapter seven contains a detailed description of the parts of the implementation that perform the actual relativistic effects. The resulting application of the project is presented in chapter eight with screen shots of different interesting scenarios together with a simple user's guide. Finally, general conclusions about the project are presented in chapter nine.

Chapter 2

The Effects

The apparent visual effects of objects moving at a relative speed close to the speed of light include an aberration effect, a headlight effect and a Doppler effect. In this chapter these effects are described.

2.1 The aberration effect

The aberration effect is a direct consequence of the speed of light being constant. If an object moves an appreciable distance in the time it takes light rays emitted from points on the object to reach the observer, the object will appear warped. The greater the distance between a point on the object and the observer, the older version of that point the observer will see. This generally means that objects moving towards the observer will appear stretched out and objects moving away from the observer will appear shortened.

A way of implementing this effect, incorporating the principle of length contradiction, can favorably be explained by a moving pinhole camera example[6]. In a pinhole camera (Figure 2.1) light rays enter through a pinhole into a box and creates a picture on the image plane. When an object moves with a velocity close to the speed of light it is subject to the special relativistic effect of length contraction and is shortened in the direction of motion by a factor of

$$\sqrt{\left(1 - \frac{v^2}{c^2}\right)} \tag{2.1}$$

In the case of the pinhole camera this means that the distance between the pinhole and image plane is shortened. Also, the camera continues to move after that a ray enters the pinhole until it reaches the image plane. Both these effects shorten the distance the light has to travel between the pinhole and the image plane. This results in light rays reaching the image plane closer to the center of the image than it would in a pinhole camera at rest. This also results in that light rays emitted from behind may be *caught* by the camera and end up on the image plane (Figure 2.2).

These effects can be summarized in a formula[12] that alters the angle θ that the incoming ray makes with the direction of motion according to the ratio between c, the speed of light and v, the speed of the camera (equation 2.2).

The equation 2.3 for calculating the new angle θ' is derived from equation 2.2 and is used in the vertex shader technique to calculate new positions for all vertices in the scene.

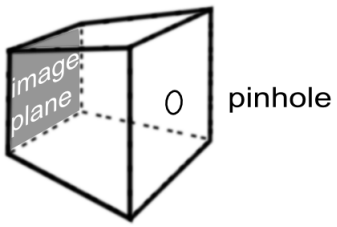


Figure 2.1: A pinhole camera.

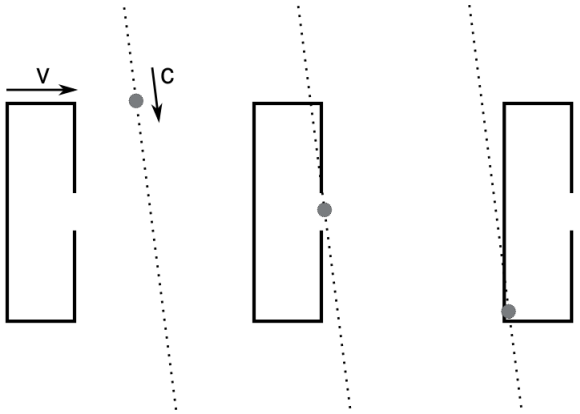


Figure 2.2: Explanation of how a photon emitted from behind a pinhole camera can end up on its image plane. v is the velocity of the camera and c is the speed of light.

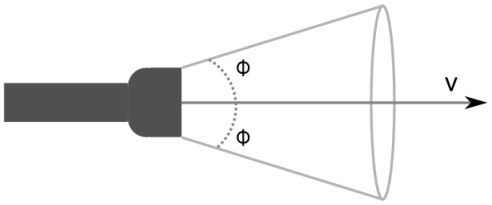


Figure 2.3: A torch that emits light in the shape of a cone.

In the pixel shader technique the problem is inverted where θ' is given and θ needs to be calculated according to equation 2.4.

$$\tan\left(\frac{1}{2}\theta'\right) = \left(\frac{c-v}{c+v}\right)^{\frac{1}{2}} \tan\left(\frac{1}{2}\theta\right) \tag{2.2}$$

$$\theta' = 2 \times \arctan\left(\sqrt{\frac{c-v}{c+v}} \tan\left(\frac{1}{2}\theta\right)\right) \tag{2.3}$$

$$\theta = 2 \times \arctan\left(\frac{\tan\left(\frac{1}{2}\theta'\right)}{\sqrt{\frac{c-v}{c+v}}}\right) \tag{2.4}$$

The relation between the angles θ and θ' is independent of the optical instrument and the same aberrational effects arise when the objects are observed with the naked eye.

One of the principles of Einstein's theory of special relativity is that there is no well-defined state of rest, this means that the same effects occur regardless of whether it is the camera or the objects that are static in the world frame. The intuitive explanation for the aberrational effects that occur when objects move close to the speed of light, observed by a non moving camera, differ from the pinhole camera model and are not discussed in this thesis because they are not considered when implementing the effect.

2.2 The headlight effect

As with the aberrational effect there are two intuitive ways of explaining the headlight effect, this time the moving object, static camera scenario is considered[1] inspired by Adam Auton's lesson five[5]. Objects moving at relativistic velocities will have the light rays emitted from them contracted into the direction of motion.

More specifically, imagine a torch that emits light in the shape of a cone in the direction of motion (Figure 2.3). At rest the angle that the cone's sides make with the pointing direction is ϕ. As the cone starts to move with a velocity v, the angle ϕ will become narrower according to

$$\sin(\phi') = \frac{\sqrt{1-\frac{v^2}{c^2}}\sin\phi}{1+\frac{v}{c}\cos\phi} \tag{2.5}$$

[1]Remember, according to the theory of special relativity there is no difference, the camera and the objects are simply moving in relation to each other regardless of which (if any) is still in relation to the world.

The amount of light emitted from the torch remains the same, only focused into a narrower cone. This leads to a increase in light intensity I according to equation 2.6. Of course, this effect is not unique to objects emitting light in a perfect cone but apply to all objects emitting light.

$$I' = \frac{\left(1 + \frac{v}{c}\cos\theta\right)^2}{\sqrt{1 - \frac{v^2}{c^2}}} \times I \tag{2.6}$$

The angle θ is the angle that the observing camera makes with the direction of motion of the object. Equation 2.6 was used when implementing the headlight effect.

2.3 The Doppler effect

The Doppler effect is a well known phenomenon that is observable in everyday life with regards to sound waves[2]. What happens with sound is that the sound waves are compressed in the direction of motion of the emitter and decompressed in the opposite direction. This results in a variance in the pitch of the sound depending on where the observer is relative the emitter.

Light waves are affected in the same way when the emitter is moving close to the speed of light resulting in a color shift, objects moving away from the observer are shifted blue and objects moving towards the observer are shifter red. At high speeds the frequency may shift outside of the spectrum visible to the human eye rendering the object practically invisible.

The color frequency λ of a relativistically moving object corresponds to

$$\lambda' = \lambda \frac{1 - \frac{v}{c}\cos\theta}{\sqrt{1 - \frac{v^2}{c^2}}} \tag{2.7}$$

The Doppler effect is not implemented in this project due to time limitations.

[2] i.e. the sound of the sirens of a moving ambulance change in frequency as the ambulance passes by the observer.

Chapter 3

Problem Description

3.1 Problem Statement

The project should produce a simulation created using Microsoft XNA where the user can travel along a predefined path manipulating the speed of the movement as well as the speed of light. The environment in which the user travels should contain a number of models provided by Coldwood Interactive. The application should be able to run on a computer system with multiple monitors. In addition to the moving point of view parts of the monitors should display a static point of view observing the moving user represented by a model.

The problem can be divided into the following three major parts.

3.1.1 Implement world environment

The environment of the simulation should consist of models provided by Coldwood Interactive. These are created in Maya and need to be imported into XNA along with their respective textures. For satisfying visual quality the application should support models with diffuse, specular, normal and dark maps. The user should be able to travel along a predefined path through the world and be able to manipulate the speed of the movement as well as the speed of the light. The speed of the movement may never reach or exceed the speed of light.

3.1.2 Implement relativistic effects

Two primary techniques are identified for implementing the aberration effect. One being altering the positions of vertices inside the vertex shader. The other being rendering the scene to a cube map and then render a screen aligned quad using a pixel shader that uses the cube map, altering what texels are mapped to what pixels according to the transformation effect. Both of these techniques should be implemented and compared with regards to usability, visual quality and performance. Additionally, if there is time, the headlight and Doppler effects may also be implemented.

3.1.3 Implement multi-monitor support

The application should run on a computer system with multiple monitors displaying two different views. One view should display the scene from the moving user's point of view

where the surrounding world is affected by the relativistic effects. The other should display a static point of view observing the moving user (which is represented by a model) in which only the user's model is affected by the effects.

3.2 Goals

The goal of the project is to produce a working interactive simulator using at least one technique of visualizing the relativistic effects that occur when moving close to the speed of light. The simulator should contain several different models and support diffuse, specular, normal and dark maps. The simulator should run on a modern computer system that has multiple monitors with a reasonable frame rate.

3.3 Purpose

The purpose of the application is to let the user experience what it is to travel close to the speed of light.

3.4 Methods

In order to test the techniques of visualizing the relativistic effects an initial environment with a camera and a few objects had to be created. Thankfully XNA is designed to facilitate creation of such environments. Still, a basic understanding of how XNA works and how it handles shaders is essential. A very useful resource for learning XNA is the many samples at the XNA website[10]. In order to implement the aberrational effect research in Special Relativity and Lorentz transformations is needed. The same basic algorithm was used for both of the techniques but implemented in different ways (see section 2 for explanation). In order to reach the desired visual quality of the models, support for additional model textures had to be implemented. The main focus was on importing the models with their respective textures correctly from Maya into the XNA environment, this was done in collaboration with the graphics department at Coldwood Interactive.

The implemented techniques were then compared with regards to the aspects presented in section 3.1.2.

3.5 Related Work

A substantial amount of work regarding visualization of relativistic effects has been done at the *Universität Tübingen*. Their *Spacetimetravel* website[7] gather a number of online version of articles written on the subject. One particularly interesting article for this project is *Through the city at nearly the speed of light*[6] by Ute Kraus and Mark Brochers. In this article an application is described where a bike ride through a virtual city at relativistic speeds is simulated, not unlike the application developed in this project.

Another application similar to the one developed in this project is *Real-Time Relativity*[11] developed by C.M. Savage, A.C. Searle, L. McCalman and M. Williamson at The Australian National University. It renders the scene using a cubemap and utilizes the GPU for calculating the relativistic effects. This application follows from C.M. Savage and A.C. Searle's earlier work *The Backlight relativistic ray tracer*[13] which is "a graphics package developed for producing photo-realistic images of relativistically moving objects"[12]. It

uses a ray-tracing technique to render images and videos without the restriction of having to do it in real time. A lot of examples using the *Backlight relativistic ray tracer* can be found on both A. Searl's *Relativistic Optics* website[13] and the *Through Einstein's Eyes* website[4].

Chapter 4

Programmable Shaders

The concept of programmable shaders play a central role in modern computer graphics and they are an obvious choice for implementing visual effects such as the relativistic ones. To achieve an understanding of what programmable shaders do and why they exist, a general understanding of the conceptual stages of the graphics rendering pipeline is crucial. In this chapter an overview of this pipeline is presented along with a general description of the graphics processing unit (GPU) that is used to execute the operations of the programmable shaders.

The XNA 3.1 framework used in this master's thesis project only supports DirectX 9.0. The programmable geometry shader was not introduced until 10.0, hence, it is not considered in this thesis.

4.1 The Graphics Rendering Pipeline

Rendering real-time graphics on modern systems is done using a pipeline architecture. This architecture is commonly used to speed up processes of variating nature. Imagine a conveyor belt with two machines doing two equally time consuming tasks each on each object, by instead using four machines doing one task each, the objects would pass through the system at half the time. Ideally, a division of a non-pipelined system into n stages lead to an increase in speed of n. The speed, or throughput, of a pipelined system is measured by its bottleneck, the stage which takes the longest to complete.

The graphical rendering pipeline consist of three conceptual stages; the application stage, the geometry stage and the rasterizer stage. The geometry and rasterizer stages are further divided into substages. Although this is the conceptual structure of the pipeline, some of the substages can be completely replaced by programmable shaders that may alter, add to, or totally replace the functionality of the substages. These are further explained in section 4.2.

4.1.1 The Application Stage

This stage is executed entirely on the general purpose CPU and the programmer has full control over what happens. The role of the application stage in the graphics rendering pipeline is to feed geometries (i.e. points, lines, triangles) to be rendered to the geometry stage. The application stage is not divided into substages in the same manner as the

geometry and rasterizer stages, however it may utilize several CPU processing cores to execute in parallel to increase performance.

4.1.2 The Geometry Stage

The geometry stage is responsible for the majority of the per-polygon and per-vertex operations[3]. The functionality of this stage could potentially be executed on the CPU, and it was, prior to the introduction of graphical processing unit's (GPU's) beginning in 1995[8]. The geometry stage is divided into the following substages; model and view transform, vertex shading, projection, clipping and screen mapping.

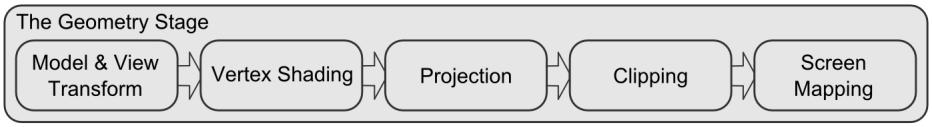


Figure 4.1: The substages of the geometry stage.

Model and View transform

A model typically undergo two transforms before being rendered, these are the model and view transforms. A transform is implemented as a 4×4 matrix and applies to the vertices and normals of a model. Originally a model is considered to be in its own *model space* before any transforms are applied. A world transform is applied to the model to position, orient and scale the model in the *world space* common to all models of the scene. It is possible for a model to have more than one world transform associated with it if it is to appear more than once in the scene, this prevents replication of the original model and is called instancing[3]. Once all models reside in the common *world space* they are transformed by the view transform into *camera space* (or *eye space*) to facilitate projection and clipping. The view transform is based on the camera viewing the scene. It rotates and positions all objects equally so that the camera ends up at the origin looking down the z-axis with its directional vectors up and right aligned to the y and x axis respectively[1].

The model and view transforms are typically applied in one step using one combined matrix, however this is not the case in the *vertex morph technique* (section 7.4) since the aberrational effect is applied on vertices in *world space*.

Vertex Shading

The next substage of the geometry stage is vertex shading. Note that this is not the programmable vertex shader described in the following section which actually is capable of a lot more than just shading. Shading is defined as the process of calculating the outputted radiance (color) along the view ray based on material properties and light sources[3]. The vertex shading substage models the appearance of each model according to their material and possible light sources shining on them. Vertices can contain a variety of data besides the position such as a color, a normal or any other data needed to compute the shading equation. The results of the vertex shading substage is then sent to the rasterizing stage to be interpolated across the pixels of the final image. Typical resulting vertex data include colors, texture coordinates and vectors (normals, tangents, bi-normals). Many of these

[1] Exact orientation of the camera differ depending on the underlying programming interface (API).

depend on whether the actual shading calculations are made in the geometry or rasterizing stage.

Projection

After the vertex shading substage the models are projected from the three dimensional *camera space* to the two dimensional *normalized device space*. This is done using a 4×4 matrix that transforms the view volume[2] into a unit cube with extreme points at (1,1,1) and (-1,-1,-1)[3]. The shape of the view volume depends on what projection method is being used. The two most commonly used methods are orthographic and perspective projection. The view volume of orthographic viewing is normally a rectangular box. This means that the transform is a combination of a translation and a scaling. The view volume of perspective viewing is a truncated triangle with a rectangle base. This is the view we are used to with our eyes, the further away an object is from the camera, the smaller it appears. The transform is called a projection since the z-coordinates are not stored in the image[3] generated, this results in a two dimensional representation of the three dimensional models.

Clipping

Clipping is the process of filtering out primitives and parts of primitives that lie outside the view volume. This substage is facilitated by the fact that the scene is already view transformed and projected because it makes the clipping process consistent; primitives are always clipped against the same unit cube[3]. Evaluating primitives that lie entirely inside or entirely outside the unit cube is trivial. Primitives that have vertices both inside and outside the cube are clipped against the cube, vertices outside are discarded and new vertices are created at the clipping plane.

Screen Mapping

Finally, the clipped primitive's x- and y-coordinates are transformed into screen coordinates. Assume that the scene should be rendered into a window with the minimum corner at (x_1, y_1) and maximum corner at (x_2, y_2) where $x_1 < x_2$ and $y_1 < y_2$ then the screen mapping is a translation followed by a scaling operation[3]. The z-coordinate is not affected by this mapping. All coordinates are then passed to the rasterizer stage.

4.1.3 The Rasterizer Stage

The next stage of the graphics rendering pipeline is the rasterizer stage. The purpose of this stage is to convert the inputted transformed and projected vertices with their associated shading data from the geometry stage into pixels on the screen[4]. Just like the geometry stage the rasterizer stage is divided into substages, these are; triangle setup, triangle traversal, pixel shading and merging.

Triangle Setup

In this stage data needed for interpolating the various shading data produced in the geometry stage is computed.

[2]The volume of the scene that is visible to the camera.

[3]The z-coordinates are rather stored in the Z-buffer, see section 4.2.

[4]More specifically to the render target, since the target of the rendering need not be the screen but actually an off screen texture.

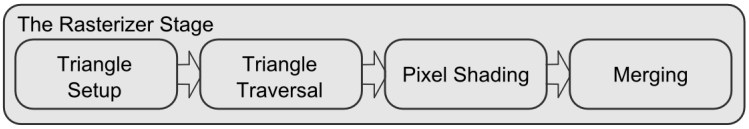


Figure 4.2: The substages of the rasterizer stage.

Triangle Traversal

For each pixel that is at least partially covered by the triangle a fragment is generated. The properties of the fragment which is to be sent to the pixel shader[5] are calculated by interpolating among the three triangle vertices. These properties include the fragment's depth as well as any shading data from the geometry stage[3].

Pixel Shading

The interpolated data passed on from the triangle traversal stage is then used to perform per-pixel shading calculations in the pixel shading stage. A lot of visual effects are dependent on the pixel shading stage, some examples are texturing, normal mapping (or *bump mapping*) and various per-pixel lighting techniques. The result that is passed on to the final substage is usually a color and possible additional data. It may also be that the pixel is discarded in the pixel shading stage.

Merging

The data passed on from the pixel shading stage is stored in rectangular buffers, these are typically color in the *color buffer*, depth in the *depth buffer* (or *Z-buffer*) and alpha in the *alpha buffer*. These buffers are used in combination in the merging stage to determine the final colors of the pixels.

4.2 The Graphical Processing Unit

Graphics cards have grown significantly over the past decade. At first they had totally fixed pipelines but with time more and more parts became programmable starting with the vertex shader and then the pixel shader and, with the introduction of DirectX 10.0 in 2006[9], an optional programmable geometry shader was added. The programmable shaders are the primary way of controlling the GPU. Shaders started out being programmable only in assembly like code, with the introduction of DirectX 8.0 in 2002 a high level language called High Level Shading Language (HLSL) started to evolve, replacing parts of the assembly instructions[9]. With the introduction of DirectX 10.0 the pipeline became virtually 100% programmable using HLSL and assembly is no longer used to generate shader code. Other similar high level shading languages that have been developed alongside Microsoft's HLSL are OpenGL's GLSL and NVIDIA's Cg. Additional languages for general purpose programming[6] using the GPU also exist, NVIDIA's CUDA is one example.

The conceptual pipeline described in section 4.1 is translated to an implementation on modern GPU's according to figure 4.3. Worth noticing is that some of the conceptual

[5] in OpenGL the pixel shader is called *the fragment shader*.

[6] Utilizing the performance boost of GPU's for executing general purpose code is usually not necessarily related to graphics rendering.

substages are fixed in their functionality, some are configurable and some are replaced by totally programmable shaders. The only advantage of using the hardware dedicated GPU instead of the software powered CPU for rendering graphics is speed, but speed is crucial[3].

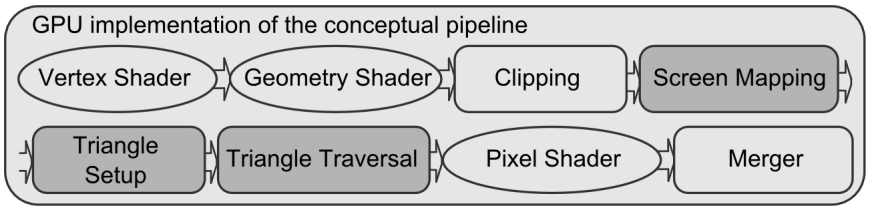


Figure 4.3: The GPU implementation of the conceptual graphics rendering pipeline in DirectX 10.0. Circular stages are fully programmable, light square stages are configurable and gray stages are fixed in their functionality.

4.3 Programmable shaders

In the GPU implementation pipeline there are a few stages missing from the conceptual graphics rendering pipeline. The programmable vertex shader has replaced the substages *Model & View Transform*, *Vertex Shading* and *Projection* from the geometry stage. The *Pixel Shading* substage of the rasterizer stage has been replaced by the programmable pixel shader. The functionality of these substages is instead left to the programmer to implement thus, giving the programmer full control. The geometry shader is an optional shader that is not supported in DirectX 9.0 which used in this project, it is therefore not discussed in this thesis.

4.3.1 Programming Shaders

Although shaders replace functional stages of the graphics rendering pipeline they need not necessarily implement that functionality. For instance, the vertex shader of the *Screen-QuadMorph* effect (further explained in section 7.3.2) contains no model or view transform but rather it just projects a screen sized quad over the entire screen. Another example is the pixel shader for the models in a pipeline using deferred shading where no colors are calculated[7].

The style of programming a shader is somewhat different from the style used when programming the CPU. The difference is a consequence of the hardware architecture. The GPU is designed to efficiently process streams of similar data such as vectors. The CPU is a serial processor, fetching and executing instructions one at a time[14].

As mentioned before there are three major languages for programming shaders, NVIDIA's Cg, OpenGL Shading Language (GLSL) and Microsoft's High Level Shading Language (HLSL). In this project HLSL was used. In HLSL effect files are created that can possibly contain several techniques and functions (shaders). This facilitates organization of different variations of graphical effects by a combination of different vertex and pixel shaders[9]. For instance the *RelativisticModelShader* implemented for rendering the aberrational effect using the vertex shader contains a total of six shaders (four vertex shaders and two pixel

[7]In deferred shading only the data needed for determining the colors are calculated. The colors are not calculated until the final stage, only on pixels that are visible.

shaders) that are combined in four different ways making up four different techniques. What technique that is used depends on if the aberrational effect is activated and whether the model contains normal mapping or not.

HLSL offers a lot of intrinsic functions that are efficiently implemented with regards to the hardware of the GPU. These include a lot of common operations used in graphics equations. Some examples of math functions are[9] *sin()*, *cos()*, *tan()*, *abs()*, *min()*, *max()*, *sqrt()* and *radians()*. A very common data type used in shader programming is the vector, there are many intrinsic vector functions, some examples are *cross()*, *dot()*, *reflect()*, *normalize()* and *length()*. Another common data type used is textures, HLSL offers intrinsic functions for querying texels of different kinds of textures, the basic ones are *tex1D()*, *tex2D()*, *tex3D()*, and *texCUBE()*, although there is a total of 24[9] functions. Intrinsic functions for matrix manipulation are also included such as *transpose()*, *determinant()* and *mul()* which is capable of multiplying matrices and vectors.

Another feature of HLSL that greatly facilitates rearranging elements of a vector is *swizzling*. This allows the programmer to quickly in one operation select any elements of a vector to form a new one. This is best explained using an example. Imagine A and B being vectors of length 4 with their components being x, y, z and w respectively. If $A = \{1, 2, 3, 4\}$ and $B = A.xwyz$ then B would equal $\{1, 4, 2, 3\}$, the "$.xwyz$" is called the swizzling operator. Swizzling can be used to form vectors of arbitrary length i.e. $A.xy = \{1, 2\}$.

HLSL also allows for flow control using *if* and *else* statements. This is used in many of the shaders developed in this project, often to decide whether a certain texture (i.e. specular map) should be added to the calculation depending on whether it is supplied with the model currently being rendered.

Chapter 5

Comparison of the implemented techniques

5.1 Theoretical performance comparison

In this section the performance of the two techniques used to visualize the relativistic aberrational effect are compared based on theoretically calculated data. Only the costs of the shader executions needed for each technique are considered. It is likely that the CPU stage (Application stage) further adds to the difference in performance between the two techniques in favor of the vertex morphing technique since the Cubemap pixel morph technique needs to render the scene to the six sides of the cube and thus need to setup six different views in each frame.

The attributes that are considered in the comparison are the number of instructions and cycles for the combined shaders used for each technique. AMD's *GPU ShaderAnalyzer*[1] is used to count the number of instructions that each shader compiles to. NVIDIA's *ShaderPerf 2*[2] is used to calculate the number of cycles required by a Geforce 8 series graphics card to execute each of the shaders used. In Table 5.1 the results for each individual shader using these programs are presented.

The two techniques utilizes parts of these shaders in different ways. To calculate the total GPU workload required by each shader equations 5.1 and 5.2 are used.

$$RMS.VS\_Morph \times numVertices + RMS.PS \times numPixels \tag{5.1}$$

Shader	Instructions	Cycles
RelativisticModelShader.VS	97	37
RelativisticModelShader.VS_Morph	216	117
RelativisticModelShader.PS	81	40
ScreenQuadMorph.VS	61	40
ScreenQuadMorph.PS	177	94

Table 5.1: Number of instructions and cycles calculated for each shader. The number of instructions are calculated using AMD's *GPU ShaderAnalyzer* and the number of cycles using NVIDIA's *ShaderPerf 2*.

$$(RMS.VS \times numVertices + RMS.PS \times 1024^2) \times 6 + SQM.VS \times 4 + SQM.PS \times numPixels \quad (5.2)$$

Equation 5.1 denotes the usage of shaders by the *Vertex morph* technique. It simply uses one vertex shader (*RelativisticModelShader.VS_Morph*) and one pixel shader (*RelativisticModelShader.PS*). These are multiplied by the amount of vertices and pixels respectively to estimate the total amount of instructions or cycles generated by the technique. Equation 5.2 denotes the usage of shaders by the *Cubemap pixel morph* technique. Since this technique renders the scene to a cube and then uses a screen aligned quad to render the final image it uses a different, bigger set of shaders compared to the *Vertex morph* technique. First, the scene is rendered once for each of the six sides of the cube using the *RelativisticModelShader.VS* vertex shader and the *RelativisticModelShader.PS* pixel shader with a fixed resolution of 1024x1024. Finally, the image is rendered using the *ScreenQuadMorph.PS* pixel shader upon a four vertex quad that is aligned to the screen by the *ScreenQuadMorph.VS* vertex shader.

Equations 5.1 and 5.2 are used together with the data outputted by *GPU ShaderAnalyzer* and *ShaderPerf 2* respectively to compare the two techniques. The comparison is done with regards to number of instructions and cycles respectively at two different screen resolutions[1] with the number of vertices ranging from 0 to 100000. The results of each technique are plotted together for each case in figures 5.1 and 5.2. Figure 5.3 presents a comparison where the number of vertices is fixed at 10000 and the number of pixels vary from 500000 to 5500000.

The exact values in these figures do not accurately describe the performance of each technique since the application stage is not considered. However, they do describe a correlation between the techniques usage of the GPU and allow for a *rough comparison* of the expected performances.

Worth noticing is that throughout the entirety of the cases the cubemap pixel morph technique is consistently more expensive than the vertex morph technique. As expected, the cost of the techniques grow linearly with increasing number of vertices and resolutions.

By comparing the gradients of figures 5.1 and 5.2 that have varying number of vertices with figure 5.3 which have varying resolutions one can predict that the cost of the techniques are much more dependent on the resolution rather than the number of vertices in the scene.

5.2 Practical comparison of the implemented techniques

5.2.1 Visual Quality

The vertex morph technique operates solely on the vertices in the scene, this causes irregular bends of the objects forms when the vertices of an object being morphed are projected too far apart on the screen (i.e. when the object is close to the camera). This is demonstrated in figure 5.5. For the same reasons textures may appear irregularly mapped, this is especially noticeable on textured planes that typically do not have that many vertices. This is demonstrated in figure 5.7. As seen in figure 5.6, the cubemap pixel morph technique does not suffer from this since it operates on the pixels of the object rendered with no morphing applied.

[1] One resolution represents a small screen (1024x768) and the other represents three bigger screens next to each other (5760x1200)

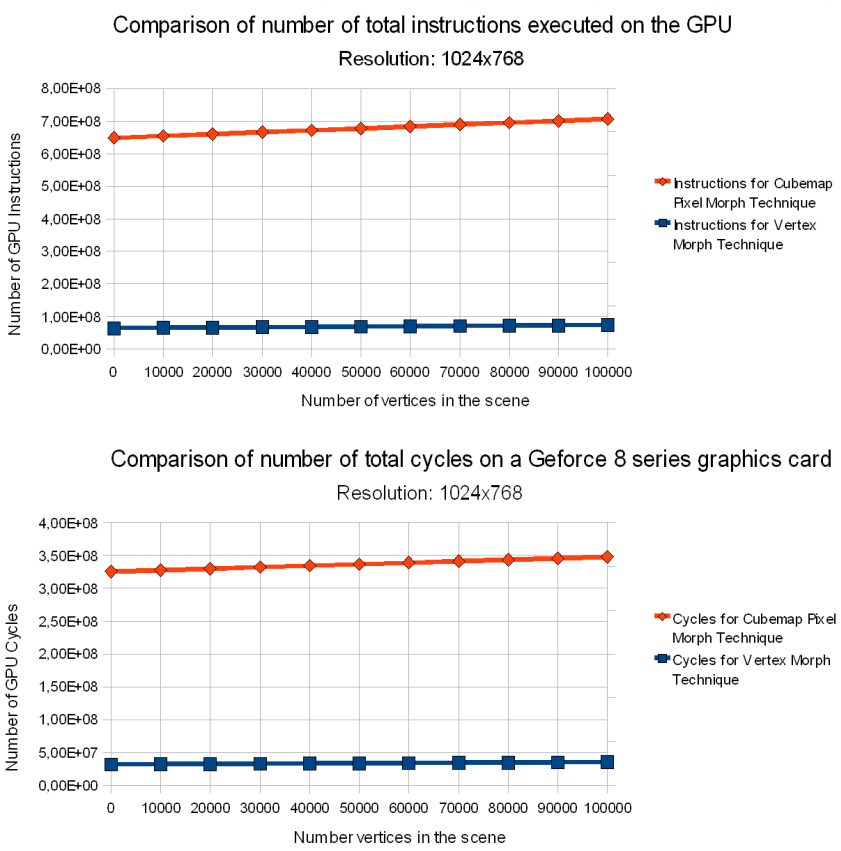


Figure 5.1: A comparison of the total number of instructions and cycles generated per frame by the shaders used by each technique at a resolution of 1024x768 and a varying number of vertices.

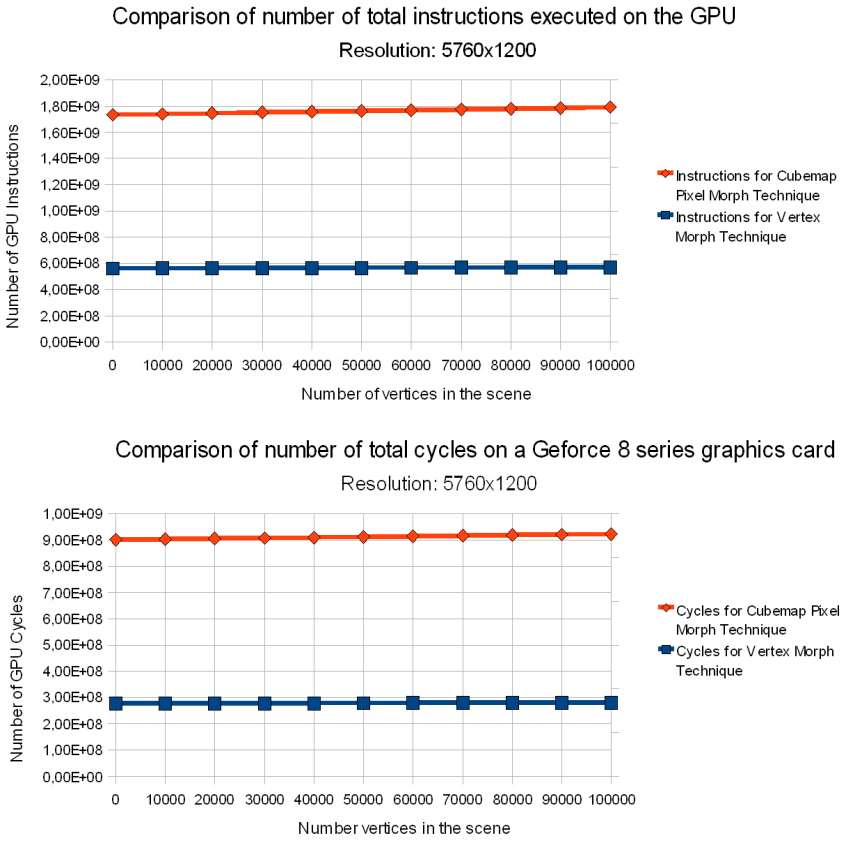


Figure 5.2: A comparison of the total number of instructions and cycles generated per frame by the shaders used by each technique at a resolution of 5760x1200 and a varying number of vertices.

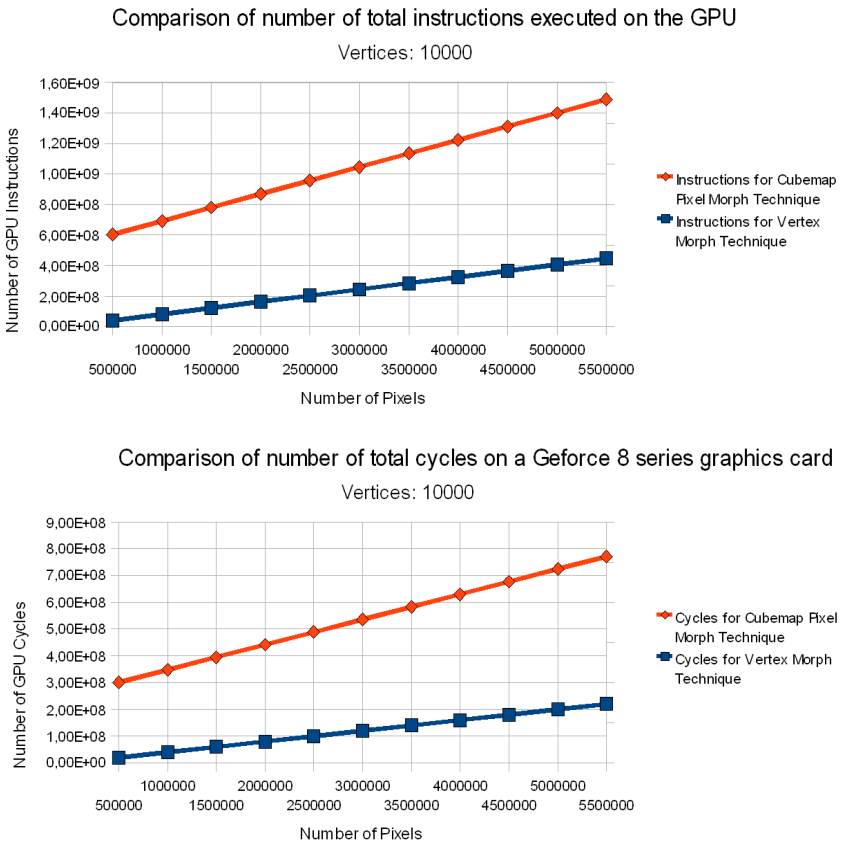


Figure 5.3: A comparison of the total number of instructions and cycles generated per frame by the shaders used by each technique in a scene with 10000 vertices with varying resolutions.

On objects with reasonably many vertices this shortcoming of the vertex morph technique is only apparent when the camera is near the object. What really distinguishes the two techniques in this matter are when large objects with few vertices such as the ground model and sky box are morphed.

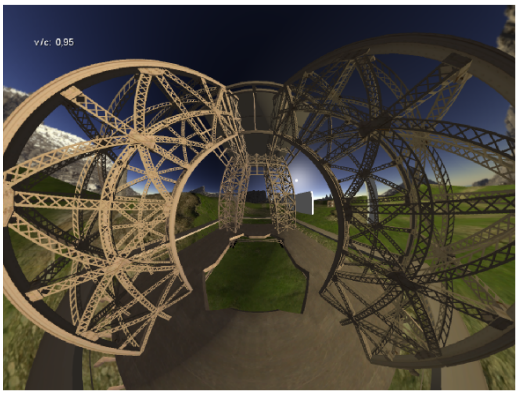


Figure 5.4: A closeup of the Eiffel tower at 95% the speed of light using the cubemap pixel morph technique.

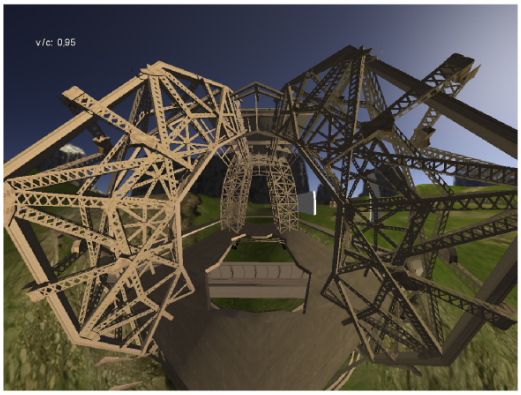


Figure 5.5: A closeup of the Eiffel tower at 95% the speed of light using the vertex morph technique.

A visual shortcoming unique to the cubemap pixel morph technique is aliasing towards the edges of the image. The degree of aliasing is dependent on the speed and viewing direction relative the moving direction. This shortcoming is a consequence of the cubemap textures being finite. As the speed of the observer approaches the speed of light the rays fetching texels from the cubemap are spread further and further apart which results in undersampling of the cubemap textures. This deficiency is most prominent when the viewport is elongated in either width or height since the longer away from the center, the further apart the texel-fetching rays will be. This is demonstrated with a view of the Eiffel tower seen at 98% of the speed of light with an aspect ratio of 3:1 in figure 5.8. Figure 5.9 show the same view using instead the vertex morph technique. This deficiency is effectively hidden by the headlight effect (section 2.2) which focuses the incoming light rays in the direction of motion darkening out the surrounding areas in which the aliasing occurs. This is demonstrated in figure 5.10.

5.2.2 Usability

Both the vertex morph and cubemap pixel morph techniques perform the morphing effects outside the application stage (section 4.1.1) and could therefore be used together with different applications with relative ease as long as the application provide the required parameters. The vertex morph technique requires in addition to the standard *model*, *view* and *projection* matrices a vector denoting the direction of motion together with a float representing the degree of morphing. The cubemap pixel morph technique require some additional information the most prominent being a cubemap texture (six textures) containing the scene surrounding the camera. The vertex morph technique replaces the rendering of all objects and need therefore be manipulated if additional visual effects are desired (i.e. shadows). The cubemap pixel morph technique requires the scene to be rendered to a cube texture but it does not interfere with the original rendering of the scene. This means that if any

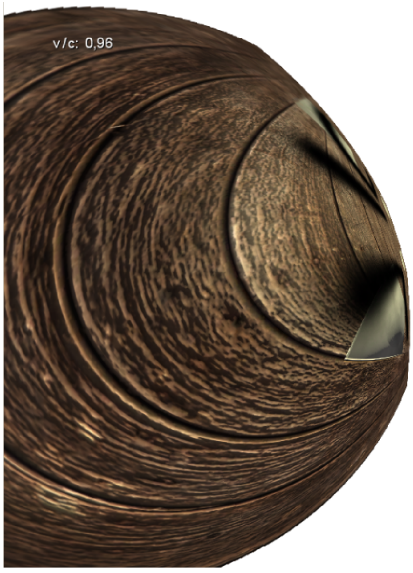


Figure 5.6: A closeup of a textured plane at 96% the speed of light using the cubemap pixel morph technique.

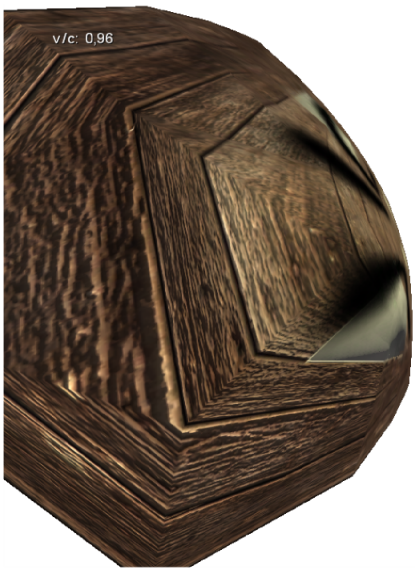


Figure 5.7: A closeup of a textured plane at 96% the speed of light using the vertex morph technique.

application would be willing to render its scene to a cubemap, the cubemap pixel morph would work. Another feature of the cubemap pixel morph technique is that it is compatible with any images, it is not limited to images generated in a computer simulation.

Due to the visual artifacts that occur when using the vertex morph technique it requires that the models have high enough vertex-resolution in relation to how close they are to the camera.

Rendering of a totally static scene

The pixel shader technique operates on a cube texture and thus requires the scene to be rendered six times per frame. However, the functionality of rendering the relativistic effects is separated from the actual rendering of the scene. This technique is totally independent from the contents of the scene and has no problems morphing objects with few vertices and high resolution textures.

Rendering a scene containing moving objects

Rendering objects that move within the scene (in addition to the observer) is relatively easy with the vertex shader technique. The vertex morphing effect is applied to the moving object and the moving direction parameter is set to the objects inverted movement relative the observer.

The cubemap pixel morph technique however requires a cubemap to be rendered for each moving object. Each cubemap is located with the observer in its center but is rotated in the inverted moving direction of the object currently being rendered. The only thing rendered from the cubemap is the object and the result is blended together with the rest of the scene. When rendering these kinds of moving objects using the cubemap pixel morph

Figure 5.8: A screenshot of the right side of the screen showing the Eiffel tower at 98% of the speed of light using the cubemap pixel morph technique.

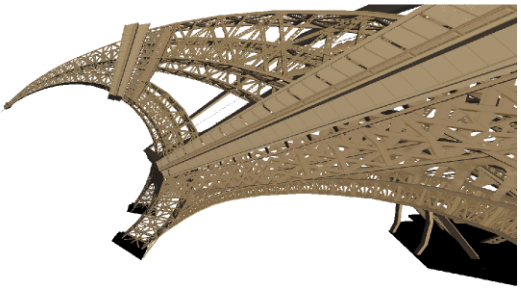


Figure 5.9: A screenshot of the right side of the screen showing the Eiffel tower at 98% of the speed of light using the vertex morph technique.

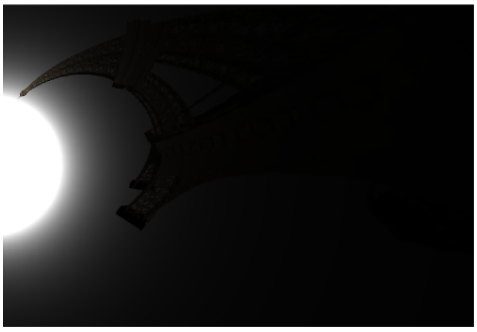


Figure 5.10: A screenshot of the right side of the screen showing the Eiffel tower at 98% of the speed of light using the cubemap pixel morph technique with the headlight effect enabled.

technique they often suffer from high aliasing, even more than demonstrated in figure 5.8.

For rendering objects moving within the scene a combination of the techniques are used, the static scene without moving objects is rendered using the cubemap pixel morph technique and the moving objects are rendered using the vertex morph technique.

5.2.3 Performance

The simulation was tested with regards to performance on the system specified in table 5.2. The frames per second (FPS) was measured using five different resolutions[2] and two different number of vertices[3]. The reasoning behind this is that the resolution and complexity of the scene are the two major factors to performance except for the computations done in the steps of the rendering pipeline which are decided by the technique used. The results of these tests can be seen in figures 5.11 and 5.12.

OS	Microsoft Windows XP
CPU	Intel(R) Core(TM)2 6600 @ 2.40GHz
RAM	2GB
GPU	ATI Radeon X1950
Resolution	3200x1200 (2 x 1600x1200)

Table 5.2: Test system specification.

The results of the performance tests indicate that the cost of the vertex morph technique is a lot more dependent on the number of vertices than the cubemap pixel morph technique. This is expected as the vertex morph technique executes the costly morph algorithm on each vertex. The difference between the no morphing case and the vertex morph technique is only the actual morph algorithm. As the resolution increases, especially when there are few vertices (figure 5.11) this difference grow smaller as the relative cost of the application stage and standard lighting rendering increases.

The cubemap pixel morph technique seem almost unaffected by the number of vertices in the scene. The performance of this technique appear to be decided by the resolution of the screen which makes sense since it performs the morphing on a per pixel level. The fluctuations in performance using this technique is a lot less than for the vertex morph. This would indicate that the main bottleneck is the application stage which, no matter the final resolution, always renders the scene without morphing six times to a render target with resolution 1024x1024 (the textures used for cube mapping).

The differences in performance of the two techniques differ from those in the theoretical analysis, although the same topology occurs (the cubemap pixel morph is the most expensive). One explanation of this could be that, as implied by the results in this section, the application stage is the bottleneck for the cubemap pixel morph technique and this stage was not considered in the theoretical analysis[4].

[2]Resolutions: 1024x768, 1280x1024, 1600x1200, 1920x1200 and 3200x1200
[3]Number of vertices: 15545 and 68678
[4]Only the GPU workload was considered in the theoretical analysis.

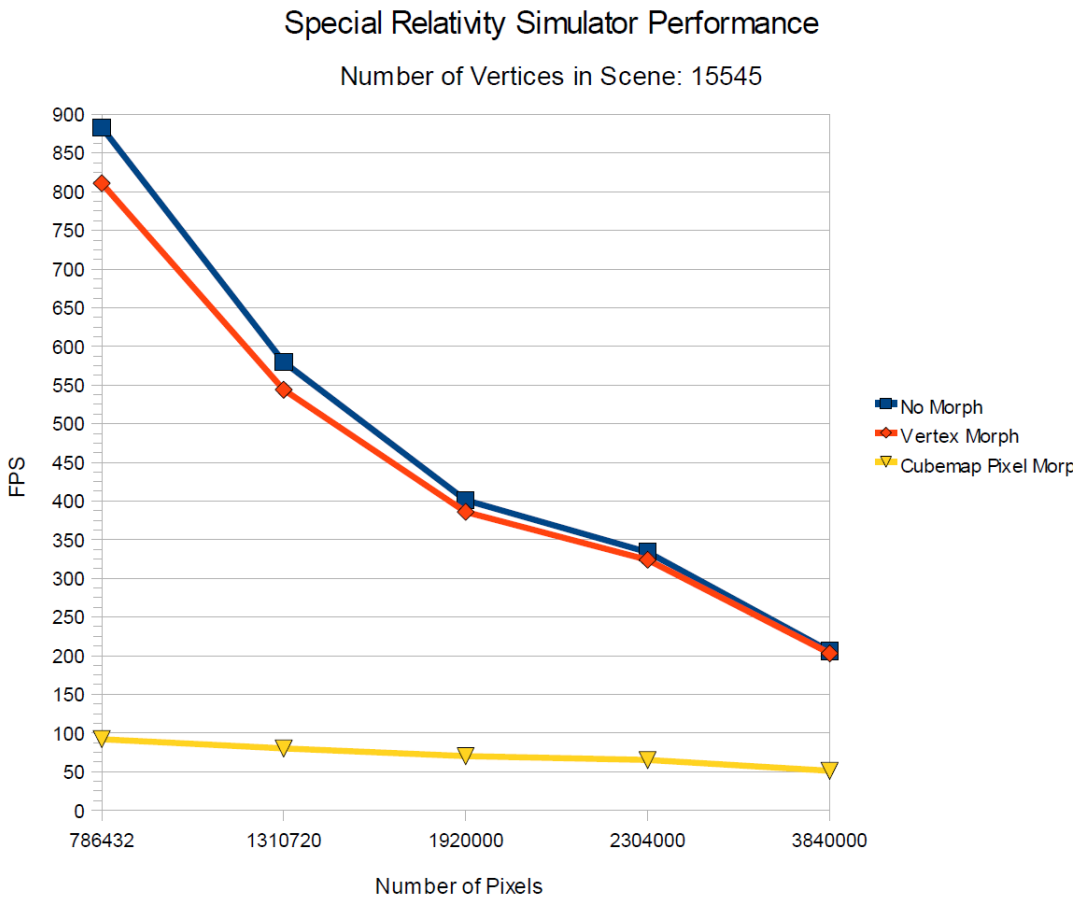


Figure 5.11: A comparison of the two techniques with regards to FPS in a scene with 15545 vertices at different screen resolutions.

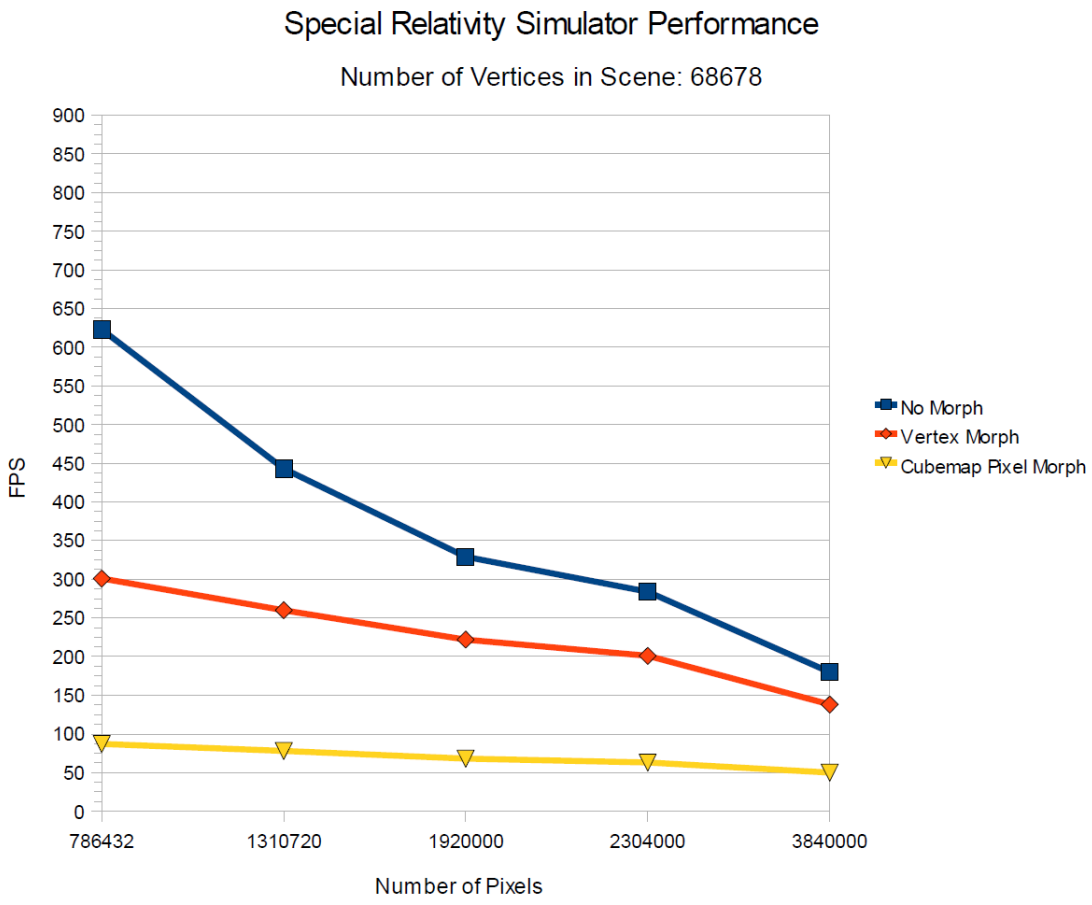


Figure 5.12: A comparison of the two techniques with regards to FPS in a scene with 68678 vertices at different screen resolutions.

Chapter 6

Accomplishment

In this chapter the process in which the work was done is described and compared to the preliminary schedule.

6.1 Preliminary Schedule

Seen below is the preliminary weekly schedule of the project. Implementation of the chosen techniques are to be finished until week 49 when the theoretical study is set to begin. The report is planned to be written concurrently to all other tasks.

37	Develop specification
38	Make preparations
39	Make preparations, meet with supervisor
40	Start at Coldwood, Familiarize with working environment (XNA)
41	Design and implement basic application
42	...
43	Research, test and choose techniques of implementing the visual effects of special relativity at the speed of light
44	...
45	Implement chosen techniques
46	...
47	...
48	...
49	Theoretical study
50	Finalize Application
51	...
52	Holiday
53	...
1	Finalize Report
2	...
3	Prepare presentation and opposition
4	Present thesis/opposition

Table 6.1: Preliminary Schedule

6.2 How the work was done

Presented in this section is an overview of how the work was done. Parts of the implementation that ended up in the final application are further explained in chapter 7 and parts that were scrapped are briefly covered in this section.

In the beginning weeks 37 through 39 mostly preparations were made and the specification document was written. In week 40 the planning document was written and the implementation phase began at Coldwood Interactive. In week 41 Coldwood supplied a basic project containing a few models; a ground, the Eiffel tower and the Turning torso. Support for custom shaders was implemented along with a per-pixel phong lighting shader. Shadows were implemented according to a basic shadowmap sample from the XNA creators club[10]. The visual quality of the shadows was not satisfactory and they were scrapped. In preparation of implementing the effects at the speed of light research was made in the area of special relativity and Lorentz transformations.

In week 42 the aberration effect (section 2.1) was implemented using the vertex shader (the *vertex morph technique*). The theory of this effect as described in chapter 2 is that the vertices in the scene are morphed towards the direction of motion of the observer. The first attempt of this morphed the vertices after they had been projected to screen coordinates using the X,Y and depth values to calculate the angle the vertices made with the optical axis. The result of this was an orb-like morph in the center of the screen that grew smaller as the speed of the observer increased. By instead using the original world coordinates of the vertices when performing the warp the correct effect was acquired. This technique is explained in further detail in chapter 7.

The cubemap pixel morph technique require the scene to be rendered to a cubemap, support for this along with the cubemap pixel morph technique was implemented in week 43. In the vertex morph technique equation 2.3 is used to calculate the new morphed angle of each vertex. In this technique however the opposite is needed, the morphed angle that points to what texel of the cubemap should be visible at a certain point on the screen. Therefore the equation used in the vertex morph technique was converted into an equation that takes the morphed angle and returns the original angle (equation 2.4). This technique is further explained in chapter 7. Additionally, this week support for diffuse, specular, normal and dark maps for models created in Maya was added. The graphics designer is supplied with the appropriate shader which is used when assigning materials to the model in Maya. The models are then loaded through a custom programmed content pipeline. The custom content pipeline is further discussed in chapter 7.

In week 44 the multi-monitor aspect of the system was considered. The first approach involved creating individual windows for each display that when maximized would entirely cover their respective displays. The functionality of this method fulfilled the specification but required some setup of each window and introduced a drop in performance since it was unable to run in true full screen[1]. This method is demonstrated in figure 6.1. Due to the mentioned deficiencies this method was scrapped. By stretching the display horizontally across all monitors in the video card settings only one big graphics adapter is visible to the application. Using this setup no consideration need to be taken as too how many monitors are used and it is possible to run the application in true full screen.

The feature of morphing one single model from a static point of view was also implemented this week. This was both done using the cubemap pixel morph and the vertex morph technique.

[1] In true full screen the back buffer and front buffer are the same size. This is optimal when swapping the pixel colors of the front buffer (screen) to those of the back buffer.

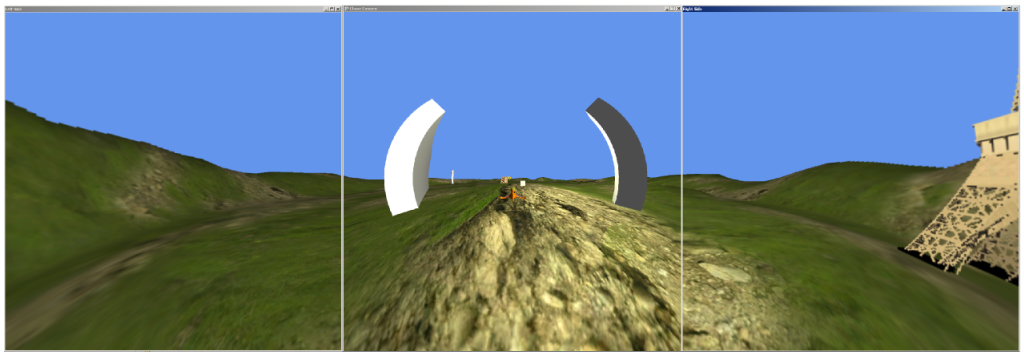


Figure 6.1: Multi-monitor support using one window for each monitor.

In week 45 the cubemap pixel morph technique was optimized in the sense that operations that could be linearly interpolated across the four vertices of the screen aligned quad were moved from the pixel to the vertex shader. This functionality include altering of the screen coordinates and calculation of the *ray to texel* (the techniques are further explained in chapter 7). This optimization resulted in a noticeable boost in performance.

The split screen functionality was also implemented this week, where some of the monitors display the moving observers view and some the view of the observer represented as a model moving in an otherwise static scene.

The headlight effect (section 2.2) was also implemented this week as a part of the cubemap pixel morph technique.

Additionally, ways of implementing the camera moving along a predefined path at various speeds were researched. The alternatives recognized included using a NURBS curve, using an animated camera or letting the user control the camera on a road using physical collisions to constrain it from wandering off. The animated camera method was chosen and implemented. XNA partially supports animations but require the content pipeline to be configured for the keyframes of the animation to be properly loaded. To achieve a smooth animation at low playback speeds the keyframes need to be interpolated. The animation would play back irregularly and different interpolating techniques were implemented.

In week 46 this issue was resolved, the problem did not lie in the interpolating step but rather in a precision error caused by the big differences in the position and direction vectors (unit length vectors were used to represent direction while the position vectors were of size 10^5). This week the application was finalized with real-time manipulation of the speeds of the observer and the light. The skysphere was also added.

Weeks 47 through 2 was spent writing the report. Concurrently to this during weeks 48 through 51 the theoretical study was performed. The theoretical study is divided into two parts. In the first part (chapter 4) the conceptual graphics rendering pipeline and the role of programmable shaders is discussed. In the second part (chapter 5) the implemented techniques were compared both theoretically with regards to performance and practically with regards to performance, visual quality and usability.

Parts of week 52 and entire week 53 was spent on holiday.

6.3 Conclusion

The models supplied by Coldwood combined with the 3D graphics facilitating nature of XNA greatly decreased the amount of time required to setup a basic environment. Two weeks were allocated for this task in the preliminary schedule, consequently, the following tasks were done two weeks prior to their preliminary date.

Researching the techniques of visualizing the effects of special relativity was scheduled for two weeks (43 and 44). This was in fact done partially in weeks 40 and 41 and resulted in a successful implementation of the vertex morph technique already in week 42. The techniques were continuously researched and implemented through weeks 42 to 45. The task of implementing the techniques required as much time as it was scheduled to, 4 weeks, however it was done three weeks earlier than planned. Due to this, the finalizing of the application which was scheduled to take place week 50 was done already in week 46.

The preliminary schedule stated that the report should have been written concurrently to all other tasks, in reality writing of the report did not start until after the implementation phase in week 47. The weeks earned in the implementation phase were spent on extending the allocated time for the theoretical study from one to three weeks. Also, more time was spent on explicitly writing the report than what was originally planned.

Chapter 7

Implementation

In this chapter the implementation of selected parts of the system is described in detail. Sections 7.1 and 7.2 describe the application and the content reader. In section 7.3 the two effect files that are used in different ways by the morphing techniques are described. Finally, the morphing techniques are described in sections 7.4, 7.5 and 7.6.

7.1 The application

The relativistic effects are all executed on the GPU. Therefore the application[1] mostly contains standard functionality needed for rendering three dimensional models. This includes i.e. loading the models through a custom content pipeline (section 7.2), handling user input and updating the camera position. Some functionality is however required by the application for the morphing techniques to work. The most prominent requirement is that a cubemap needs to be created for the cubemap pixel morph technique.

7.1.1 Creation of a cubemap

The cubemap is created by setting the cameras field of view to 90° and rotating it to each of the six axis aligned directions (positive/negative X, Y and Z). The procedure is described in figure 7.1.

7.2 The custom content pipeline

XNA comes equipped with a basic yet configurable content reader. In order to use animations and textures (additional to the diffuse texture) of the *.fbx* models created in Maya this content reader needs to be extended.

Each model used in the system is loaded through the custom content reader. To prepare possible animation clips of the models for usage in the application this content reader iterates over all animation keyframes and stores them in an array which is passed along with the model.

[1] The *application* as used here refers to the same functionality span as that of the application stage in section 4.1.1

1.	Create **projection** matrix with a 90° field of view.
2.	Calculate normalized direction of camera.
3.	Save the old depth buffer.
4.	For each of the six faces of the cube:
4.1.	Calculate **look-at point** and **up** direction for the original camera if it would be rotated towards the cube face.
4.2.	Create **view** matrix using the cameras original **position** and newly calculated **look-at point** and **up** vector.
4.3.	Set the **view** and **projection** matrices of the cubemap camera to those newly created.
4.4.	Set the current render-target to the correct face of the cubemap texture.
4.5.	Render the scene using the cubemap camera.
5.	Reset the render-target and depth buffer.

Figure 7.1: Creation of the cubemap.

A similar approach is used for textures, in addition to adding the textures to the model object flags are set for each found texture. These flags are later used within the *RelativisticModelShader* shader by dynamic branching to decide what textures exist and should be rendered for each model.

7.3 The effect files

Effect files is a feature of HLSL that facilitates the organization of shaders and combination of shaders into *techniques*. An effect file can contain any number of shaders and any number of combination of these. The shaders of this project are arranged in two effect files, *RelativisticModelShader.fx* and *ScreenQuadMorph.fx*.

7.3.1 RelativisticModelShader.fx

This effect file contains the shaders that are applied to the models of the scene. The pixel shaders are used to render models with an arbitrary set of textures lit by a single light source. The vertex shaders contained in this file may either place the models without morph in the scene or morph them according to the inputted morphing factor (the vertex morph technique). Below two shaders are described, the vertex shader that performs the morph effect of the vertex morph technique (figure 7.3) and the pixel shader that applies lighting and various textures to the model (figure 7.4). The shaders described here do not support normal mapping, however, there are normal mapping variants of all shaders within the RelativisticModelShader effect file. Input data shared by all shaders of the RelativisticModelShader effect file is presented in figure 7.2. The HLSL data type entries [*data type*][*number*] denote vectors of length *number* with elements of type *data type*. [*data type*][*number* x *number*] instead denote matrices with the specified dimensions.

7.3.2 ScreenQuadMorph.fx

This effect file contains only one technique (one vertex shader and one pixel shader). The purpose of this technique is to render the scene to a screen aligned quad using a cubemap

Data type	Description
float4x4	World matrix.
float4x4	View matrix.
float4x4	Projection matrix.
float	The morphing factor. (velocity/speed of light)
float3	The direction the camera is moving.
float3	Position of the light source.
float4	Ambient light color.
float4	Diffuse light color.
float4	Specular light color.
float	Specular power.
float	Specular intensity.
texture	Diffuse texture.
texture	Specular, normal and dark map textures.
	(These textures may or may not be set)

Figure 7.2: Input data shared by all shaders of the RelativisticModelShader effect file.

and adding possible morphing (the cubemap pixel morph technique) and headlight effects. The vertex shader that sets up the screen aligned quad (figure 7.6) and the pixel shader that renders the scene using a cubemap (figure 7.7) are described below.

7.4 The vertex morph technique

This is one of the two techniques for achieving the visual aberrational effects that occur when traveling close to the speed of light. As the name implies this technique applies the effect to the vertices of the object subjected to the effect. A parameter common to both techniques is the morphing factor which is calculated as v/c^2. This technique uses the RelativisticModelShader vertex shader (as described in figure 7.3) in which the vertices are positioned so that the shape of the model is consistent with the current morphing factor. The vertices are then passed to a standard pixel shader that performs phong shading using an arbitrary set of object specific textures to render the object.

7.5 The cubemap pixel morph technique

This is the second technique for achieving the visual aberrational effects when traveling close to the speed of light. Opposed to using the vertex positions to morph the shape of the object the morphing is done as a post process using a cubemap. This technique is divided into two steps. In the first step a cubemap is created in the application stage as described in figure 7.1. In the second step the cubemap together with the morphing factor[3] is passed to a pair of shaders (the vertex and pixel shaders of the *ScreenQuadMorph* effect file) that renders a screen aligned quad with a texture representing the morphed scene. These shaders are described in figures 7.6 and 7.7 accordingly.

[2] v/c - the velocity of the observer over the speed of light.
[3] For a complete list of inputted parameters see figure 7.5

	Input
float4	The vertex position in object space.
float3	The normal of the vertex in object space.
float2	The texture coordinates for the diffuse, specular and normal map.
float2	The texture coordinates for the dark map.
	Output
float4	The vertex position in screen space.
float2	The texture coordinates for the diffuse, specular and normal map.
float2	The texture coordinates for the dark map.
float3	The direction to the light source from the vertex.
float3	The direction to the camera from the vertex.
float3	The vertex normal in world space.
	Execution
1.	Calculate position in world space by multiplying the input position with the world matrix.
2.	Extract the cameras position from the view matrix by multiplying the three first elements of the third row with its transpose.
3.	Calculate the direction from the camera to the vertex.
4.	Calculate the angle the direction to the vertex makes with the direction of the camera.
5.	Calculate the angle after morph according to equation 2.3.
6.	Calculate the new position of the vertex using the morphed angle.
6.1	Calculate the amount to rotate the original position by taking the difference between the original and the morphed angle.
6.2	Calculate the axis to rotate about by calculating the cross product of the direction to the vertex from the camera and the direction of the cameras motion.
6.3	Build a quaternion using the rotation amount and the axis to rotate about.
6.4	Calculate a ray from the camera to the new morphed position of the vertex using the quaternion.
6.5	Position the vertex at the new position.
7.	Multiply the vertex position by the view and projection matrices.
8.	Calculate the direction to the light source from the vertex.
9.	Calculate the direction to the camera from the vertex.
10.	Transform the normal to world coordinates by multiplying it with the world matrix.
11.	Output texture coordinates without change.

Figure 7.3: Description of the RelativisticModelShader.VS_Morph shader.

	Input
float4	The vertex position in world space.
float2	The texture coordinates for the diffuse, specular and normal map.
float2	The texture coordinates for the dark map.
float3	The direction to the light source from the vertex.
float3	The direction to the camera from the vertex.
float3	The vertex normal in world space.
	Output
float4	The final color of the pixel.
	Execution
1.	Calculate the diffuse factor by taking the maximum of the dot product of the normal and direction of light and zero.
2.	Calculate the diffuse lights contribution to the final color by multiplying the diffuse color with the diffuse factor.
3.	Calculate the direction of the reflected light by reflecting the incoming light about the normal of the vertex.
4.	Calculate the specular factor by taking the maximum of the dot product of the reflected light and the direction to the camera and zero.
5.	Calculate the specular lights contribution to the final color by multiplying the the specular factor to the power of the inputted specular power with the specular light color.
6.	If the specular texture is set:
6.1	*True*: Look up the value in the specular texture and multiply is with the specular contribution.
6.2	*False*: Multiply the specular contribution with the color of the diffuse texture and the inputted specular intensity.
7.	Calculate the color of the pixel according to: $(diffusecontribution + ambientcontribution) \times diffusetexturecolor + specularcontribution$
8.	If the dark map texture is set:
8.1	*True*: Multiply the color of the pixel with the value of the dark map.
8.2	*False*: Do nothing.
9.	Output the final color of the pixel.

Figure 7.4: Description of the RelativisticModelShader.PS shader.

Data type	**Description**
float	The morphing factor. (velocity/speed of light)
float	The cameras rotation about the y-axis from the direction of motion in degrees.
float	The aspect ratio.
float	The field of view.
float	Percentage of the screen that the quad should fill from left to right.
boolean	A boolean indicating whether the headlight effect is active.
cube texture	The cubemap texture.

Figure 7.5: Input data shared by both shaders of the ScreenQuadMorph effect file.

	Input
float4	The vertex position in world space.
float2	The texture coordinates of the vertex.
	Output
float4	The position of the vertex in screen space.
float2	The texture coordinates of the vertex.
float3	The ray from the camera to the texel of the cubemap denoted by the texture coordinates.
	Execution
1.	Position the vertices of the quad so that the quad fills the screen from left to right according to the inputted percent value.
2.	Invert the y-axis of the texture coordinates (x, y) and alter them so that the center texel is at coordinates (0, 0) taking into account the field of view (FOV) and aspect ratio: $textureCoordinates = (textureCoordinates - (0.5, 0.5)) \times \tan(FOV/2) \times (aspectRatio, -1)$
3.	Determine the direction of a ray from the camera to the texel corresponding to the texture coordinates of the vertex. This information will be interpolated to the pixel shader which in turn will receive rays to each texel that should be displayed according to the current aspect ratio and field of view.
4.	If the parameter that denotes the cameras rotation about the y-axis is not zero:
4.1	*True*: Rotate the ray about the y-axis according to the inputted degree value.
4.2	*False*: Do nothing.

Figure 7.6: Description of the ScreenQuadMorph.VS shader.

	Input
float3	The ray from the camera to the texel of the cubemap denoted by the texture coordinates.
	Output
float4	The final color of the pixel.
	Execution
1.	Calculate the angle the inputted ray makes with the direction of motion (which is pointing straight down the positive z-axis).
2.	Rotate the ray to point at another texel according to the inputted morphing factor.
2.1	Calculate the new morphed angle according to equation 2.4.
2.2	Calculate the amount to rotate the original ray by taking the difference between the original and the morphed angle.
2.3	Calculate the axis to rotate about by calculating the cross product of the direction of the ray and the positive z-axis.
2.4	Build a quaternion using the rotation amount and the axis to rotate about.
2.5	Calculate the new morphed direction of the ray using the quaternion.
3.	Look up the color of the texel in the cubemap using the morphed ray.
4.	If the inputted headlight flag is set: *True*: Apply the headlight effect according to equation 2.6. *False*: Do nothing.
5.	Output the color of the pixel.

Figure 7.7: Description of the ScreenQuadMorph.PS shader.

7.6 The headlight effect

The headlight effect as described in section 2.2 is implemented as a part of the *ScreenQuad-Morph* pixel shader as described in figure 7.7. It is not available when using the vertex morph technique. The reasons for this is that the pixel morph technique is the primary technique used when rendering the scene and that the necessary θ angle needed for the headlight equation (equation 2.6) is already calculated per pixel in this technique.

Chapter 8

Results

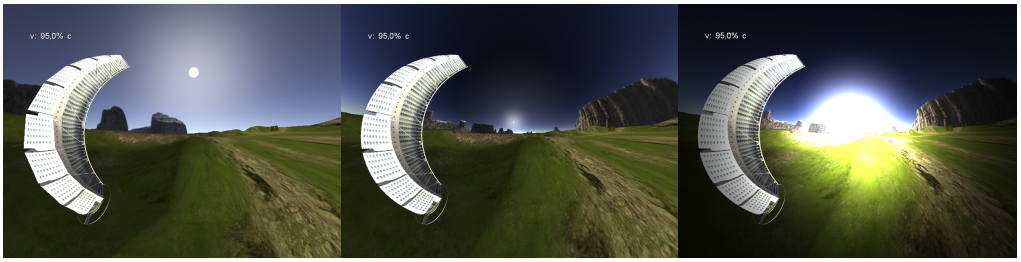
The result of this Master's Thesis project is the simulator application where the user may experience the visual effects that occur when traveling close to the speed of light. This chapter contains a simple user's guide and screenshots of the application.

8.1 User's Guide

The movement of the observer can be controlled in two modes. In one the observer follows a predefined path and the morphing factors is automatically calculated depending on the speed. In the other, the observer is flying freely in the world and the morphing factor is manually set. The keybindings for both of these modes along with some general keybindings are presented in table 8.1.

8.2 Screenshots

In figure 8.1 a view of the Turning Torso is displayed at 95% the speed of light. The three pictures are produced using the vertex morph technique and the cubemap pixel morph with and without the headlight effect respectively.



(a) Vertex morph technique. (b) Cubemap pixel morph technique. (c) Cubemap pixel morph technique with the Headlight effect ON.

Figure 8.1: The famous Turning Torso observed at 95% the speed of light generated using various techniques.

	General Keybindings
Alt + Enter	Toggle full screen.
B	Toggle cubemap pixel morph technique.
V	Toggle vertex morph technique.
G	Toggle visibility of cubemap.
NumPad7	Toggle headlight effect.
NumPad1,2,3	Rotate looking direction when using the cubemap pixel morph technique.
NumPad8	Toggle observer movement modes.
O	Toggle split screen modes.
NumPad4,5,6	Position split screen delimiter.
	Keybindings for predefined path movement.
S	Increase observer speed.
D	Decrease observer speed.
X	Increase the speed of light.
C	Decrease the speed of light.
	Keybindings for free fly movement.
S	Increase observer speed.
A	Decrease observer speed.
N	Step through morphing factor values.
Arrows	Control direction of observer.

Table 8.1: Keybindings.

In the example seen in figure 8.3 the observer is passing through a hollow cube-like object (figure 8.2) accelerating from 0 to 99% of the speed of light. Worth noticing is that in the last picture the observer is positioned entirely outside of the cube on the other side of it.

In figure 8.4 the cubemap generated in the cubemap pixel morph technique is visualized using different colors for each face. Worth noticing is that the orange colored back face becomes visible at high speeds.

In figure 8.5 a picture is taken with a camera moving at 95% the speed of light through the Eiffel tower. The aberrational effects make it look as if the camera is just outside the building although, at the time when the picture was taken the camera is positioned directly beneath the Eiffel tower.

In figure 8.6 the screen is divided into two parts. On the right part the space ship that moves perpendicular to a static camera is photographed and morphed using the vertex morph technique. On the left the view of the ship is displayed in which the cubemap pixel morph technique is used. The ship is moving at 95% the speed of light.

In figure 8.7 a view of a moving space ship is presented where the ship is moving perpendicular to the camera at 90% the speed of light and is morphed using the cubemap pixel morph technique with the headlight effect activated.

In both of these figures the space ship appear rotated away from the camera, this is a consequence of the aberration effect.

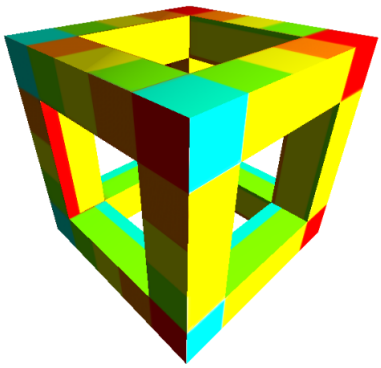
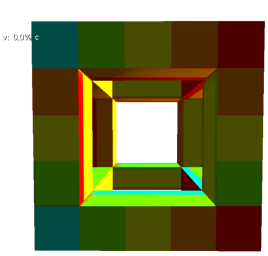
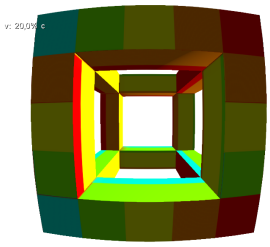


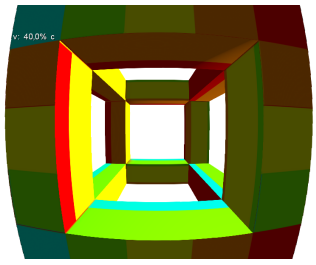
Figure 8.2: The hollow cube-like object.



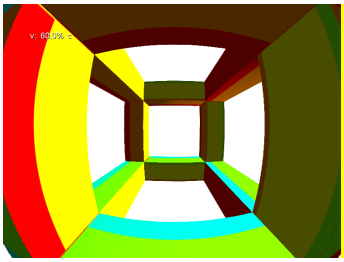
(a) v = 0% of c



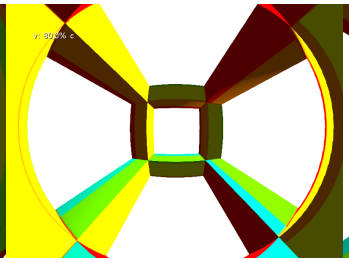
(b) v = 20% of c



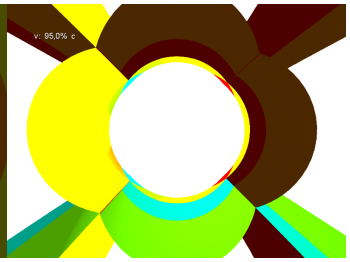
(c) v = 40% of c



(d) v = 60% of c



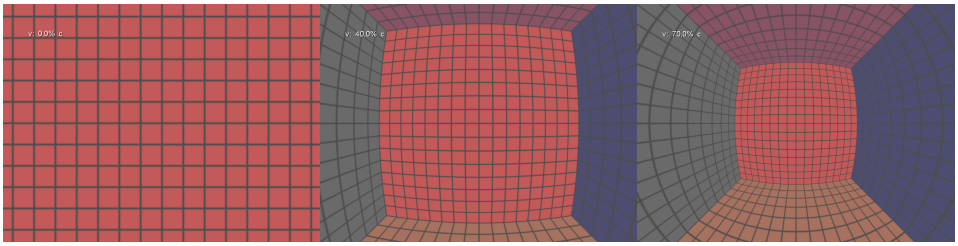
(e) v = 80% of c



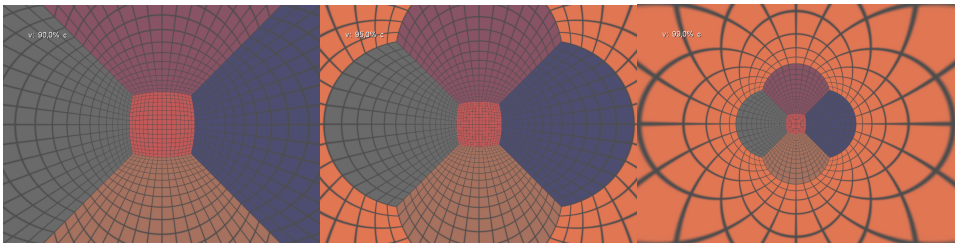
(f) v = 95% of c

(g) v = 99% of c

Figure 8.3: Passing through a hollow cube by accelerating from 0% to 99% of the speed of light.



(a) v = 0% of c (b) v = 40% of c (c) v = 70% of c



(d) v = 90% of c (e) v = 95% of c (f) v = 99% of c

Figure 8.4: A collection of images of how texels are chosen from the cubemap at various velocities ranging from 0% to 99% of the speed of light using the cubemap pixel morph technique.

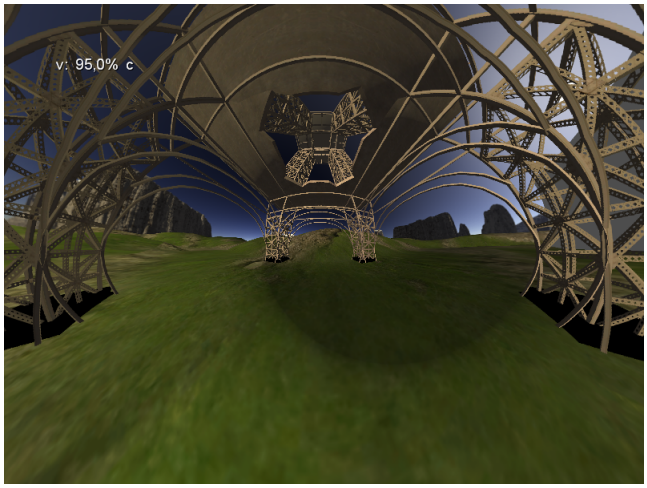


Figure 8.5: A picture take with a camera moving at 95% the speed of light passing through the Eiffel tower.

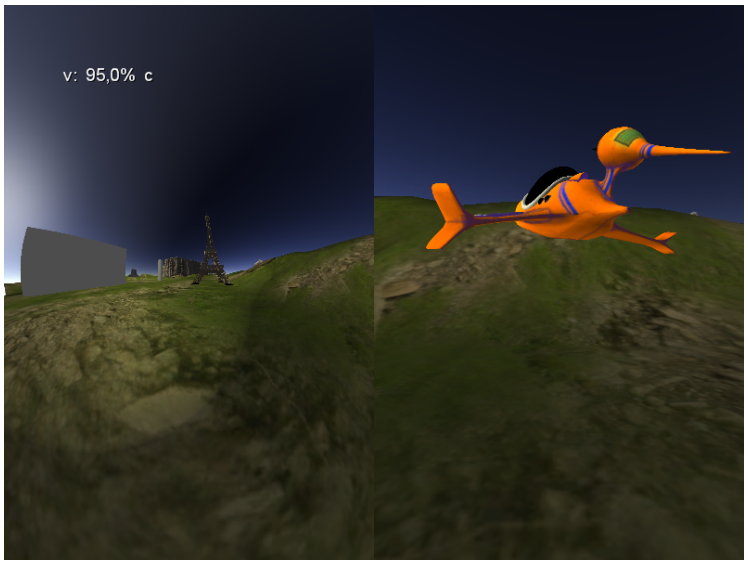


Figure 8.6: A view of the split screen mode. The view of the ship that moves with a speed of 95% the speed of light is seen on the left morphed using the cubemap pixel morph technique. On the right the moving ship is displayed using the vertex morph technique.

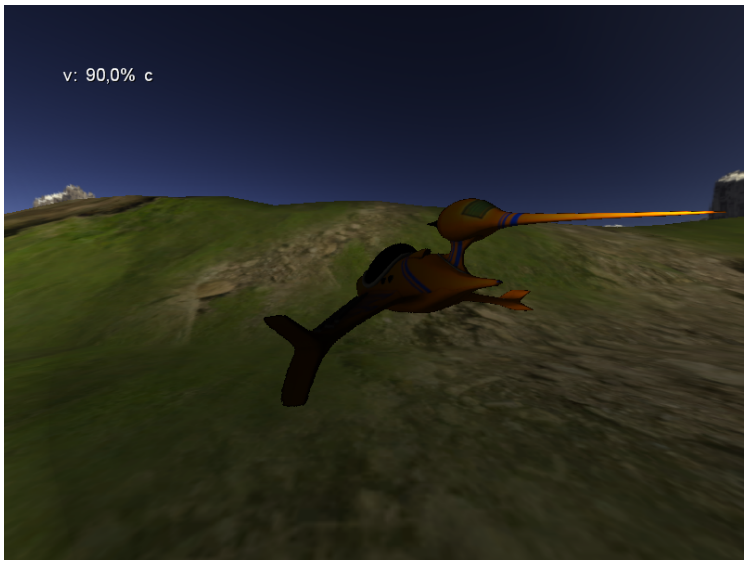


Figure 8.7: The cubemap pixel morph technique is used to morph the ship at 90% the speed of light with the headlight effect turned ON.

Chapter 9

Conclusions

The task of visualizing the apparent forms of object moving close to the speed of light in a computer simulation goes hand in hand with the constant evolving field of computer graphics. Although, the fundamental equations remain the same, the next evolutionary step could be to use ray tracing where each ray can be modeled to move at relativistic speeds. This method can of course be used today for pre-rendered images but we are not quite there yet when it comes to real-time applications.

In this project two techniques are implemented that supplement each other well in terms of visual quality relative their performance. So well in fact that both are used in the final application for different scenarios. The vertex morph technique is a lot more efficient and maybe if combined with the functionality of the geometry shader its visual quality could be increased to match that of the cubemap pixel morph technique without compromising too much of the performance. The theory behind this is to increase the vertex density at points of objects in time when the visual artifacts described in section 5.2.1 would occur.

9.1 Limitations

The aberrational effect of relativistic travel has been the main focus of this thesis, overshadowing the headlight and Doppler effects. This is mainly because this effect was the focus of the problem description but also because it is the most complex to implement. The headlight effect is of course fully implemented but only in the cubemap pixel morph technique. The Doppler effect however was left out from the implementation. The equation for the Doppler effect (equation 2.7) calculates a new color frequency λ' according to the angle θ and current frequency λ of the pixel. This requires the color frequency to be mapped to a color. There is no intuitive way of doing this without inputting an actual color scheme which would be used to look up first what frequency a given color maps to and second to find what color the newly calculated frequency maps to.

9.2 Future work

Besides from implementing the Doppler effect there are many improvements that can be made to the actual application stage of the system. The system is designed primarily to visualize the relativistic effects in relatively small scenes. To better support additional objects and features the system would benefit from a more object oriented design.

There are many ways in which the simulation could be improved, for instance adding more objects, adding moving objects, support for sound including the appropriate Doppler effects, displaying more parameters related to special relativity (such as mass, length contradiction, current time etc.), more cameras/view ports and alternative input methods.

Chapter 10

Acknowledgments

I would like to thank my external supervisor Jakob Marklund and all the other guys at Coldwood Interactive for giving me the opportunity to perform this Master's Thesis project and making it a pleasant experience. I especially want to thank Leif Holm for being my go-to guy and helping me decide how to shape the application.

I would also like to thank my internal supervisor Thomas Johansson at Umeå Universitet for helping me with the structure of the project and for giving me feedback on the report.

References

[1] *AMD GPU ShaderAnalyzer (http://developer.amd.com/gpu/shader/Pages/default.aspx, (visited 2009-12-14)).*

[2] *NVIDIA ShaderPerf 2 (http://developer.nvidia.com/object/nvshaderperf_home.html (visited 2009-12-14)).*

[3] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering, Third Edition.* A K Peters, Ltd., 2008.

[4] P. Altin, F. Bennett, M. Hush, C. Savage, and A. Searle. Through einstein's eyes. http://www.anu.edu.au/Physics/Savage/TEE/site/tee/home.html (visited 2009-11-30).

[5] A. Auton. Warp. http://www.adamauton.com/warp/ (visited 2009-11-30).

[6] U. Kraus and M. Borchers. Fast lichtschnell durch die stadt (online version: http://www.spacetimetravel.org/tuebingen/tuebingen.html (visited 2009-11-30)). *Physik in unserer Zeit*, 2005.

[7] U. Kraus and C. Zahn. Space time travel. http://www.spacetimetravel.org/ (visited 2009-11-30).

[8] P. Lilly. From voodoo to geforce: The awesome history of 3d graphics, http://www.maximumpc.com/article/features/graphics_extravaganza_ultimate_gpu_retrospective?page=0,9 (visited 02-12-09). *MAXIMUMPC (http://www.maximumpc.com/)*, 2009.

[9] Microsoft. Msdn. http://msdn.microsoft.com/ (visited 07-12-09), 2009.

[10] Microsoft. Xna creators club. http://creators.xna.com (visited 2010-01-15), 2010.

[11] C. M. Savage, A.C. Searle, L. McCalman, and M. Williamson. Real-time relativity. http://realtimerelativity.org/ (visited 2009-11-30).

[12] C.M. Savage and A.C. Searle. Visualizing special relativity. http://www.anu.edu.au/physics/Searle/paper2.pdf (visited 2009-11-30).

[13] A. Searle. Relativistic optics at the anu. http://www.anu.edu.au/physics/Searle/ (visited 2009-11-30), 2009.

[14] A. H. Watt and F. Policarpo. *Advanced game development with programmable graphics hardware.* A K Peters, Ltd., 2005.